A runtime verification system for Software Defined Networks

Victor S. Altukhov
Lomonosov Moscow State University
Moscow, Russia
Email: victoralt@lvk.cs.msu.su

Eugene V. Chemeritskiy
Applied Research Center
for Computer Networks
Moscow, Russia
Email: tyz@lvk.cs.msu.su

Vladislav V. Podymov
Lomonosov Moscow State University
Moscow, Russia
Email: valdus@yandex.ru

Vladimir A. Zakharov
Lomonosov Moscow State University
Moscow, Russia
Email: zakh@cs.msu.su

***Abstract*—We present a software toolset VERMONT (VERifying MONiTor) for runtime checking the consistency of Software Defined Network (SDN) configurations with formally specified invariants of Packet Forwarding Policies (PFPs). One of the main task of network engineering is to provide such a loading of SDN switches with forwarding rules as to guarantee its compliance with the PFP. VERMONT provides some automation to the solution of this task. Being installed in line with the control plane it 1) observes state changes of a network by intercepting the exchange of messages and commands between network switches and SDN controller, 2) builds an adequate formal model of a whole network, and 3) checks every event, such as installation, deletion, or modification of rules, port and switch up and down events, against a set of PFP invariants. Before retransmitting a network updating command to a switch VERMONT anticipates the result of its execution and checks the requirements of PFP. If a violation of some PFP invariant is detected then VERMONT blocks the change, alerts a network administrator, and gives some additional information to elucidate a possible source of an error. In this paper we discuss both mathematical and engineering issues of our toolset. We begin with defining a formal model of SDN and a formal language for PFP specification. After presenting the main algorithms used in VERMONT for SDN model building, model checking, and model modification, we describe the structure of VERMONT and the functionality of its components. Finally, we demonstrate the results of our experiments on the application of VERMONT to a real-life network.**

Keywords*—*runtime verification, formal specification, model checking, software defined network, controller, switch, packet forwarding relation, Binary Decision Diagram, network update

I. Introduction

Runtime verification is an approach to computing system analysis and verification based on extracting information, checking required properties and possibly reacting on the violation of some requirements in the course of system execution. Like bounded model checking it is intermediate between complete formal verification of a program and its testing. Runtime verification can be used for many purposes, such as security or safety policy monitoring, debugging, testing, verification, validation, profiling, fault protection, behavior modification (e.g., recovery), etc.

Runtime verification has some advantages over complete formal verification. It avoids the complexity of traditional formal verification techniques, such as model checking and theorem proving, by analyzing only one or a few execution traces. Usually a runtime verifier works directly with the actual system; thus, it scales up relatively well and gives more confidence in the results of the analysis. Moreover, runtime verification can be performed even when there is no access to the source code of the computing systems to be verified. On the other hand, a runtime verifier operates with formally specified properties of system behaviour and uses formal methods for analyzing program executions presented as traces, snapshots, etc. This allows one to achieve a far more substantial understanding of system behaviour as compared with normal testing. No wonder that these nice features of runtime verification make this approach much favor in using it for verification and analysis of the behaviour of reactive systems, such as network protocols.

In this paper we present a VERifying MONiTor (VERMONT) which is a toolset for runtime verification of Software Defined Networks (SDNs) against formally specified invariants of Packet Forwarding Policies (PFP). VERMONT is intended for preventing SDN controllers from sending incorrect network updating commands to SDN switches. Speaking figuratively, VERMONT is a fully automatic "watchdog" for SDNs.

The paper is organized as follows. In Section 2 we briefly outlook the key principles of SDN paradigm, the purpose and functionality of SDN components — forwarding rules, reconfiguration commands, switches, and a controller, — and discuss the known approaches and techniques used in SDN verification. In Section 3 and 4 we introduce a relational formal model for SDN configurations and present a formal language for PFP specifications. In Section 5 we discuss three main tasks to be solved for runtime verification of SDNs, namely, model building, model checking, and model updating, and describe in some details the algorithms used in VERMONT for the solution of these tasks. The runtime verification toolset VERMONT, its structure and functionality are described in Section 7. And, finally, in Conclusion we demonstrate some results of our experiments with VERMONT and compare our toolset with other SDN verification tools.

II. Software Defined Networks

Until recently all computer-based telecommunication networks have been built out of special-purpose hardware components (routers, switches, firewalls, gateways, etc.). These devices run a hierarchy of distributed algorithms to provide such functionality as topological discovering, routing, traffic monitoring and balancing, access control, etc. Networks of this kind are managed through a set of complex heterogeneous interfaces used to configure separately the network devices. This is a complex and error-prone activity which becomes the most severe obstacle in the development of novel network technologies such as data centers and cloud computing.

To cope with these principal difficulties a new kind of network architecture — Software Defined Networks (SDN) — has emerged some years ago [1]. The most remarkable distinguished feature of SDN architecture is that the data plane and the control plane of a network are separated from each other. We use OpenFlow [2] as the most developed standard for SDNs to give a brief overview of basic principles of SDN structure and functionality. SDN *switches* are connected with each other via their *ports* through *datapath channels*. Only *data packets* commuted by SDN switches are transmitted via datapath channels. Every switch processes packets that enter its ports; packet processing is guided by a *flow-table* — a set of *packet forwarding rules*. In general case a switch may have a whole pipeline of flow-tables. A forwarding rule includes a *pattern*, an *instruction*, a *priority*, a *counter*, and a *time-out*. A packet consists of a *header* and a *payload*, but only its header is taken into account when the packet is processed by a switch. When a switch receives a packet it matches its header against the patterns of forwarding rules. If several different rules match a packet then the rule with the highest priority is triggered. If no rules match a packet then a *default rule* is invoked. When a rule is triggered its instruction takes effect. An instruction is a list of *actions*. Typical actions include forwarding a copy of the packet out of one of the ports on the switch, dropping the packet, forwarding the packet to the controller for a in-depth analysis and processing, or rewriting some header fields. As soon as a rule completes processing of a packet it increments the counter. A packet forwarding rule is removed from the table at the expiration of its timeout.

SDN switches are managed by a centralized *controller* which is a general purpose machine capable of performing arbitrary computation. SDN controller communicates with every switch via a *control channel*. Through this channel the controller keeps track of the current configuration of the networks by receiving *messages* from switches and configures the network by sending *commands* to switches. The received messages include all packets forwarded to the controller, statistics (the number of packets processed by specified rules), notification about the removing of certain rules from flow tables, and switch status information. By sending commands to switches the controller is able to add, delete and modify certain packet forwarding rules in flow tables, to request some information from a switch (configuration, capabilities, statistics of the activity of certain rules), or to send definite packets from specified ports of a switch. More details on the concept of SDN and its implementation can be found in [2].

The main advantage of this network architecture is that programmers are able to control the behaviour of the whole network by installing, disabling, and modifying the packet forwarding rules in the flow tables of the switches. Therefore, SDNs can both simplify existing network applications and serve as a platform for developing new ones (see [3]). As soon as the concept of SDN emerged, a number of projects on the development of languages and tools for SDN programming have been launched: Frenetic [4], Maestro [5], Procera [6], Nettle [7]. But for SDNs as for any information processing system bugs remain problematic. A wide range of requirements may be imposed upon a communication network to guarantee its correct, safe and secure behaviour. While interacting with the switches the controller must provide a *Packet Forwarding Policy* (PFP) — a desirable communication between the nodes of a network. Certain packets have to reach their destination, whereas some other packets have to be dropped. Certain switches are forbidden for some packets, whereas some other switches have to be obligatorily traversed. Loops are not allowed. The software applications that operate on a SDN controller and manage a network have to provide such a loading of flow tables with forwarding rules as to guarantee compliance with a given PFP.

A wide diversity of communication networks that differ in their purpose, structure, characteristics together with endless variety of possible PFP requirements make the development of universal system for translating PFP requirements into effective network managing programs impossible in the nearest future. SDN programming will most likely follow the same way as traditional software engineering: high-level SDN programming languages will be developed to facilitate the work of network managers, and a broad range of software tools will be built to provide the designing of correct and efficient SDN control applications. These tools require certain formal techniques for the solution of such well known problems as software compilation, debugging, testing, simulation, etc.

One of the problems to be solved in our intention to guarantee the correctness of SDN control is that of *SDN verification against a PFP*: given a formally specified PFP Φ, a formal model of SDN M, and an initial network configurations $\mathcal{N}$ check that all runs of M on N satisfy Φ. The solution is to develop a toolset which could be able 1) to check correctness of a separate application operating on the controller w.r.t. a specified PFP, 2) to check consistency of PFPs implemented by the whole set of applications, and 3) to monitor and check correctness and safety of the entire SDN.

Formal methods have been widely used for verification of traditional networks and network protocols (see [8], [9], [10]), but the interest to verification issues in telecommunication networks burst as the SDN concept appeared. Since the first paper on this topic [11] a number of approaches and techniques have been introduced and implemented. Roughly, they can be divided into two main classes: control plane verification and data plane verification.

Control plane verification focuses on testing and verifying SDN controllers and their software applications. For this purpose model checking, theorem proving, static analysis and their combinations are used. Canini et al. [12] have proposed a symbolic execution and model checking based NICE framework for catching bugs which works by exploring symbolically all possible code paths in SDN program modeled as a finite state transition system. But experiments showed

that this approach is efficient only for rather small networks; therefore, NICE was extended to perform concoling testing to reduce the number of missed bugs [13]. Model checking approach based on data state and network state abstractions has been used in [14] for verification of SDN controllers. But the most remarkable paper in this line of research is [15]. In this paper Ball et al. presented the first system for verifying correctness of SDN program on all admissible topologies and for all possible sequences of network events. First-order logic is used to specify admissible network topologies and desired network-wide invariants, verification procedure is based on classical Floyd-Hoare-Dijkstra deductive approach supported by an automated theorem prover Z3.

As for SDN data plane verification, various approaches have been proposed to solve this task. At the beginning the researchers designed only static checking tools which take snapshots of network configurations, regard them as finite state transition systems, and check their properties (reachability analysis, loop detection, etc.) off-line using various model checking techniques — BDD-based model checking (FlowChecker [16]), SAT solving (Anteater [17]), manipulations with DNFs (Hassel [18]). At the next step various real-time dynamic checking tools have been proposed: VeriFlow [19], NetPlumber [20], AP-Verifier [21]. VeriFlow tool verifies network invariants — e.g., lack of access control violations, absence of routing loops, blackholes, etc. — in real time and presents a diagnostic report in case of a violation. NetPlumber tool uses a novel header space analysis for performing a real time PFP checking. The authors of AP-Verifier reduce the set of predicates representing patterns of packet forwarding rules to a set of atomic predicates that is provably both minimum and unique, which can be used to dramatically improve the computation of network reachability.

The verification systems of next generation — VeriFlow, NetPlumber, AP-Verifier — are real-time (runtime) checkers; they are not only able to check formal models of SDNs against formal PFP specifications, but to modify promptly SDN models in response to such events as flow table update by reconfiguration commands, expiration of forwarding rules' timeouts, etc. To achieve high speed of model modification the authors of these runtime verification systems turn to the explicit (enumerative) representations of SDN models and avoid using symbolic computations. As a result, such systems are able to check only restricted class of PFP requirements and they suffer from state explosion effect.

As for PFP specification languages, almost all verification systems mentioned above use temporal logics CTL or LTL for this purpose. The only known exception is NetPlumber; its PFP specification language is based on regular expressions. We think this choice is not adequate to the problem of SDN data plane verification, since both temporal logics and regular expressions are more suitable to express the properties of process evolutions proceeded in time, whereas SDN configurations are static objects whose semantics can be defined not in terms of computations, but by means of packet forwarding relations.

VERMONT is also a SDN data plane runtime verification system. Its distinguished features are as follows.

1) In contrast with VeriFlow, NetPlumber, and AP-Verifier, it uses BDDs for representation of SDN formal models and symbolic computations for their checking. BDD-based techniques makes our system far less sensitive to the size of SDN configurations to be checked.
2) Unlike FlowChecker, it is supplied with procedures for fast modification of SDN models and may be used for runtime ("dynamic") verification. We show that these procedures work rather efficient for SDN models represented by BDDs.
3) Its PFP specification language is based on the fragment of second order logic. This formal language is far more expressive than those used the verification systems referred above.

In Conclusion we show that VERMONT may find an intermediate position among the SDN data plane verification systems. Its performance exceeds that of "static" SDN data plane verifiers (FlowChecker, Anteater, Hassel) and remains only little behind the speed of such "dynamic" verifiers as FlowChecker and NetPlumber, while its PFP specification language encompasses all properties of SDN configurations that can be checked by all other verification systems.

III. Network Model

In this Section we define a relational formal model of SDN configurations which is used in VERMONT for SDN data plane runtime verification. In this model all components of SDN configuration are presented either by Boolean vectors, or by binary relations on the sets of Boolean vectors. This formal model of SDN has been presented in [22]; it has been used also in [23] for the study of Network Updating Problems.

Packet header is a Boolean vector $\mathbf{h} = (h_1, h_2, \ldots, h_N)$. All headers have the same length N and the set of all packet headers is denoted by $\mathcal{H}$. Components of a header $\mathbf{h}$ are denoted by $\mathbf{h}[i],\ 1 \le i \le N$. The length of packet headers may vary depending on network protocols, types of network, etc; e.g. the IPv4 packet header consists of 160 bits but optional fields for flags, tags, counters can be added.

Switch port is a Boolean vector $\mathbf{p} = (p_0, p_1, \ldots, p_k)$. Its components are denoted by $\mathbf{p}[i],\ 0 \le i \le k$. If $\mathbf{p}[0] = 1$ then $\mathbf{p}$ is an input port, otherwise it is an output port. All switches in the network are assumed to be identical and have the same number of ports. The set of all ports, input ports, and output ports of a switch is denoted by $\mathcal{P}$, $\mathcal{IP}$, and $\mathcal{OP}$, respectively. The output port $drop = (0, 0, \ldots, 0)$ is viewed as a drop port; at arriving to this port the packets are dropped. The output port $octrl = \langle 0, 1, 1, \ldots, 1\rangle$ is the control output port; at arriving to this port the packets are sent to a controller. The input port $ictrl = \langle 1, 1, 1, \ldots, 1\rangle$ is the control input port; only commands and messages from the controller come to this port.

All network switches are enumerated. The name of every switch is a Boolean vector $\mathbf{w} = (w_1, w_2, \ldots, w_m)$. Its components are denoted by $\mathbf{w}[i],\ 0 \le i \le m$. The set of such vectors is denoted by $\mathcal{W}$.

Let $\mathbf{h} \in \mathcal{H},\ \mathbf{p} \in \mathcal{P},\ \mathbf{w} \in \mathcal{W}$. Then a pair $\langle \mathbf{h}, \mathbf{p}\rangle$ is called a *local packet state*, a pair $\langle \mathbf{p}, \mathbf{w}\rangle$ is called a *network point*, and a triple $\langle \mathbf{h}, \mathbf{p}, \mathbf{w}\rangle$ is called a *packet state*. The set of all packet states is denoted by $\mathcal{S}$. Given a packet state $\mathbf{s} = \langle \mathbf{h}, \mathbf{p}, \mathbf{w}\rangle$ we denote its components by $\mathbf{s}.hd$, $\mathbf{s}.pt$, and $\mathbf{s}.sw$ respectively.

A *header pattern* is a vector $\mathbf{z} = (\sigma_1, \sigma_2, \ldots, \sigma_N)$, where $\sigma_i \in \{0, 1, *\},\ 1 \le i \le N$. A *port pattern* is a vector $\mathbf{y} = (\delta_1, \delta_2, \ldots, \delta_k)$, where $\delta_i \in \{0, 1, *\},\ 1 \le i \le k$. Patterns are used for the selection of appropriate rules from flow tables as well as for the updating of packet headers.

In our model of SDN we consider two types of *actions* defined in OpenFlow protocol [2]: forwarding actions $OUTPUT(\mathbf{p})$, where $\mathbf{p} \in \mathcal{OP}$, and header modification action $SET\_FIELD(\mathbf{z})$, where $\mathbf{z}$ is a header pattern. An *instruction* is any finite sequence of actions.

A *rule* is a tuple $\mathbf{r} = \langle(\mathbf{z}, \mathbf{y}), \alpha, \ell\rangle)$, where $\mathbf{z}, \mathbf{y}$ are header and port patterns, α is an instruction, and a positive integer ℓ is a *priority* of the rule. A *flow-table* is a pair $tab = (D, \beta)$, where $D = \{r_1, r_2, \ldots, r_n\}$ is a set of forwarding rules and β is a default instruction. A switch applies rules from its flow-table to those packets which arrive to the input ports of a switch. If all rules from the set D are inapplicable to a packet then the default instruction β takes effect. Usually in practice β just sends the packets to the SDN controller. The set of all possible flow-tables is denoted by Tab.

As opposed to SDN models introduced in [12], [24], our model deals with paths in the data plane routed by forwarding rules (per flow model) rather than with individual packets that traverse a network of switches (per packet model). Therefore, the semantics of the SDN model is defined in terms of packet forwarding relations on packet states and points. These relations are specified by Quantified Boolean Formulae. To capture the effect of patterns in forwarding rules we use two auxiliary functions $U_\sigma(u, v)$ and $E_\sigma(u)$, where $\sigma \in \{0, 1, *\}$, and u, v are Boolean variables, such that

- if $\sigma = *$, then $U_\sigma(u, v)$ is $u \equiv v$ and $E_\sigma(u)$ is 1,
- if $\sigma \in \{0, 1\}$, then $U_\sigma(u, v)$ and $E_\sigma(u)$ are $v \equiv \sigma$.

An action $a{=}OUTPUT(\mathbf{p}_0)$ sends packets (not changing their headers) to the output port $\mathbf{p}_0$. It computes the relation

$$R_a(\langle\mathbf{h}, \mathbf{p}\rangle,\ \langle\mathbf{h}', \mathbf{p}'\rangle) = \mathbf{h} = \mathbf{h}'\ \wedge\ \mathbf{p}' = \mathbf{p}_0$$

on the set of local packet states $\mathcal{H} \times \mathcal{P}$.

An action $b = SET\_FIELD(\mathbf{z})$ uses a pattern $\mathbf{z} = (\sigma_1, \ldots, \sigma_N)$ to modify headers of packets: a bit $\mathbf{h}[i]$ in a header doesn't change if $\mathbf{z} = *$, otherwise it is changed to $\mathbf{z}[i]$. This action computes the relation on the set $\mathcal{H} \times \mathcal{P}$:

$$R_b(\langle\mathbf{h}, \mathbf{p}\rangle,\langle\mathbf{h}', \mathbf{p}'\rangle){=}\bigwedge_{i=1}^{N} U_{\sigma_i}(\mathbf{h}[i], \mathbf{h}'[i]) \wedge \bigwedge_{i=1}^{k} (\mathbf{p}[i] \equiv \mathbf{p}'[i])\ .$$

An instruction α computes the relation R_α which is a sequential composition of the relations that correspond to its actions. If α is empty then a packet by default have to be dropped, i.e. sent to the port $drop$. Therefore, we assume that every instruction always ends with a forwarding action.

A forwarding rule $r = (\langle\mathbf{z}, \mathbf{y}\rangle, \alpha, \ell)$ applies the instruction α to all packets whose port and header match the patterns $\mathbf{y} = \langle\delta_1, \ldots, \delta_k\rangle$ and $\mathbf{z} = \langle\sigma_1, \ldots, \sigma_N\rangle$. Its effect is specified by the relation R_r on the set of local packet states $\mathcal{H} \times \mathcal{P}$

$$R_r(\langle\mathbf{h}, \mathbf{p}\rangle,\langle\mathbf{h}', \mathbf{p}'\rangle){=}PRC_r(\langle\mathbf{h}, \mathbf{p}\rangle) \wedge R_\alpha(\langle\mathbf{h}, \mathbf{p}\rangle, \langle\mathbf{h}', \mathbf{p}'\rangle)\ ,$$

where $PRC_r(\langle\mathbf{h}, \mathbf{p}\rangle) = \bigwedge_{j=1}^{N} E_{\sigma_j}(\mathbf{h}[j])\ \wedge\ \bigwedge_{i=1}^{k} E_{\delta_i}(\mathbf{p}[i])$ is a *precondition* of the rule r.

The semantics of a flow-table $tab = (D, \beta)$, where $D = \{r_1, r_2, \ldots, r_n\}$, is specified by a binary relation as follows. Let n be the highest priority of the rules from tab. For every $i,\ 1 \le i \le n$, denote by tab^i the set of rules from tab which have priority i: $tab^i = \{r = (\langle\mathbf{z}, \mathbf{y}\rangle, \alpha, i)\ :\ r \in tab\}$. Then define recursively (from n down to 1) the pairs of relations R^i_{tab} and B^i_{tab} as follows:

$$R^n_{tab} = \bigvee_{r \in tab^n} R_r,\ B^n_{tab} = \bigvee_{r \in tab^n} PRC_r;$$

$$R^i_{tab} = \{(\langle\mathbf{h}, \mathbf{p}\rangle,\langle\mathbf{h}', \mathbf{p}'\rangle) : \text{there exists } r \text{ in } tab^i \text{ such that}$$
$$\langle\mathbf{h}, \mathbf{p}\rangle \notin B^{i+1}_{tab} \text{ and } (\langle\mathbf{h}, \mathbf{p}\rangle,\langle\mathbf{h}', \mathbf{p}'\rangle) \in R_r\}),$$

$$B^i_{tab} = B^{i+1}_{tab} \vee \bigvee_{r \in tab^i} PRC_r.$$

Since the missed packets are managed by the default rule β, we introduce also the predicate

$$R^0_{tab}(\langle\mathbf{h}, \mathbf{p}\rangle,\langle\mathbf{h}', \mathbf{p}'\rangle) = \neg B^1_{tab}(\mathbf{h}, \mathbf{p}) \wedge R_\beta(\langle\mathbf{h}, \mathbf{p}\rangle,\langle\mathbf{h}', \mathbf{p}'\rangle).$$

Finally, $R_{tab} = \bigvee_{i=0}^{n} R^i_{tab}$; it means that every packet arrived at some port of the switch is either processed by the rule of the highest priority that matches the local state of the packet, or it is managed by the default rule β of the flow-table.

Network topology is completely defined by a *packet transmission relation* $T \subseteq (\mathcal{OP} \times \mathcal{W}) \times (\mathcal{IP} \times \mathcal{W})$. Since wired networks admit only point-to-point connections, T is an injective function. Network points that are involved in the relation T are called *internal network points*; others are called *external network points*. We denote by In and Out the sets of all external input and external output points respectively. External points of a switch are assumed to be connected to outer devices (hosts, servers, gateways, etc.) that are out of the scope of the SDN controller. Packets enter a network through the input points and leave a network through its output points.

When a set of switches $\mathcal{W}$ and a topology T are fixed then a *network configuration* is a total function $Net :\ \mathcal{W} \to Tab$ which assign flow-tables to the switches of the network. The semantics of a network at a configuration Net is specified by the *1-hop packet forwarding relation* R_{Net} on the set of (global) packet states S as follows:

$R_{Net}(\langle\mathbf{h}, \mathbf{p}, \mathbf{w}\rangle, \langle\mathbf{h}', \mathbf{p}', \mathbf{w}'\rangle)$ holds iff

- either $(\langle\mathbf{h}, \mathbf{p}\rangle, \langle\mathbf{h}', \mathbf{p}'\rangle) \in R_{Net(\mathbf{w})}$, $\mathbf{w} = \mathbf{w}'$, and $\langle\mathbf{p}', \mathbf{w}\rangle \in Out$ (a packet is forwarded to an outer device connected to an external output port $\mathbf{p}'$ of a switch $\mathbf{w}$),
- or there exists a port $\mathbf{p}''$ such that $(\langle\mathbf{h}, \mathbf{p}\rangle, \langle\mathbf{h}', \mathbf{p}''\rangle) \in R_{Net(\mathbf{w})}$ and $(\langle\mathbf{p}'', \mathbf{w}\rangle, \langle\mathbf{p}', \mathbf{w}'\rangle) \in T$ (a packet with possibly changed header is forwarded to an output port $\mathbf{p}''$ and then delivered via a communication channel to an input port $\mathbf{p}'$ of a switch $\mathbf{w}'$).

A relational formal model of SDN configuration Net is a triple $M_{Net} = (R_{Net}, In, Out)$.

Network configurations alter at the expiry of forwarding rules' time-outs, at the shutting down or failure of links, ports, or switches, and by the *network updating commands* received from the controller. OpenFlow protocol [2] includes network updating commands of the following types:

- $add(\mathbf{w}, r)$ to install a forwarding rule r in the flow-table of a switch $\mathbf{w}$;
- $del(\mathbf{w}, \langle \mathbf{z}, \mathbf{y} \rangle, \ell)$ to remove rules from the flow-table of a switch $\mathbf{w}$: a rule $r = (\langle \mathbf{z}', \mathbf{y}' \rangle, \alpha, m)$ is uninstalled iff $m = \ell$ and the pattern $\langle \mathbf{z}', \mathbf{y}' \rangle$ of the rule matches the pair $\langle \mathbf{z}, \mathbf{y} \rangle$;
- $mod(\mathbf{w}, \langle \mathbf{z}, \mathbf{y} \rangle, \beta, \ell)$ to modify the rules in the flow-table of a switch $\mathbf{w}$: if pattern $\langle \mathbf{z}', \mathbf{y}' \rangle$ of the rule $r = (\langle \mathbf{z}', \mathbf{y}' \rangle, \alpha, m)$ matches the pair $\langle \mathbf{z}, \mathbf{y} \rangle$ and $m = \ell$ then the instruction α in such rule is changed to the instruction β.

As a network updating command is delivered via a control channel to a switch it fires and changes the flow-table of the switch by installing, removing or modifying the appropriate forwarding rules. Formally, we write $com(Net)$ for the new configuration obtained at the execution of a network updating command com on a configuration Net.

IV. Specification Language

PFPs refer to properties of network configurations at some stages of the SDN behaviour. These properties mostly concern the paths routed in a network by packet forwarding rules. We choose first-order logic with two variables and a transitive closure operator (FO^2[TC] in symbols) to specify the properties of network configurations. Two variables are sufficient for FO[TC] to embed in it the most of specification languages used in formal verification such as CTL, LTL, PDL, μ-calculus; our verification system is able to cope with more variables but at the sacrifice of certain loss in performance. Our fragment of FO^2[TC] includes only three basic predicates R, I, and O to denote 1-hop packet forwarding relation and the sets of incoming and outgoing packet states. Now we consider this PFP specification language in some more details.

Let $Var = \{X, Y\}$ be a set of two variables that are evaluated over the set $\mathcal{S} = \mathcal{H} \times \mathcal{P} \times \mathcal{W} = \{0,1\}^{N+k+m}$ of packet states. A *packet state specification* is any Boolean formula φ constructed from a set of Boolean variables $X[j]$ and $Y[j]$, $1 \leq j \leq N+k+m\}$, and connectives $\neg, \wedge, \vee$. By means of these formulae it is possible to express relationships between the pairs of packet states.

A PFP specification language $\mathcal{L}$ is the smallest set of expressions which satisfies the following requirements:

1) if φ is a packet state specification then $\varphi \in \mathcal{L}$;

2) if $X', X'' \in Var$ then $R(X', X''), I(X'), O(X'')$ are in $\mathcal{L}$;

3) if $\psi(X, Y)$ is a formula in $\mathcal{L}$ and it includes exactly two distinct free variables then $TC(\varphi(X, Y)) \in \mathcal{L}$;

4) if ψ_1 and ψ_2 are formulae in $\mathcal{L}_1$ and $X \in Var$ then the formulae $(\neg\psi_1)$, $(\psi_1 \wedge \psi_2)$, $(\psi_1 \vee \psi_2)$, $(\exists X\ \psi_1)$, and $(\forall X\ \psi_1)$ are in $\mathcal{L}$.

A *PFP specification* is any closed formula in $\mathcal{L}$.

The semantics of $\mathcal{L}$ is defined as follows. Let Net be a network configuration, and $\mathbf{s} = \langle \mathbf{h}, \mathbf{p}, \mathbf{w} \rangle$ and $\mathbf{s}' = \langle \mathbf{h}', \mathbf{p}', \mathbf{w}' \rangle$ be a pair of packet states. Then

1) $M_{Net} \models R(X, Y)[\mathbf{s}, \mathbf{s}']$ iff $(\mathbf{s}, \mathbf{s}') \in R_{Net}$;

2) $M_{Net} \models I(X)[\mathbf{s}]$ iff $\langle \mathbf{p}, \mathbf{w} \rangle \in In$;

3) $M_{Net} \models O(X)[\mathbf{s}]$ iff $\langle \mathbf{p}, \mathbf{w} \rangle \in Out$;

4) $M_{Net} \models TC(\varphi(X, Y))[\mathbf{s}, \mathbf{s}']$ iff there exists a finite non-empty sequence of packet states $\mathbf{s}_0, \mathbf{s}_1, \ldots, \mathbf{s}_n$ such that $\mathbf{s}_0 = \mathbf{s}$, $\mathbf{s}_n = \mathbf{s}'$, and $Net \models \varphi[\mathbf{s}_i, \mathbf{s}_{i+1}]$ holds for every $i, 0 \leq i < n$.

The satisfiability relation for other formulae in $\mathcal{L}$ is defined straightforward as in the first-order logics.

Some simple examples show that $\mathcal{L}$ is rather expressive to formalize PFPs.

1) No loop-holes are reachable from the outside of the network:

$$\neg\exists X\ (I(X) \wedge \exists Y\ (TC(R(X, Y)) \wedge TC(R(Y, Y)));$$

2) None of the switches processes packets from both flows $flow_1$ and $flow_2$:

$$\neg\exists X\ (\exists Y\ (flow_1(Y) \wedge TC(R(Y, X))) \wedge \exists Y\ (flow_2(Y) \wedge TC(R(Y, X)))),$$

where $flow_1$ and $flow_2$ are Boolean formulae which specify the aforesaid packet flows.

There are several reasons to explain our choice of FO^2[TC] for PFP specification language. In the most papers that study verification problem for SDN (see [11], [17], [19], [20]) the authors use temporal logics (LTL or CTL) for PFP specification language. This choice is explicable when per-packet abstraction is concerned since the movement of a packet may be viewed as a process evolving in time. But inasmuch as our model has a per-flow abstraction level, temporal logics become inadequate formalism. As far as we are interested in the relationships between packet states and routes in the network configurations, FO^2[TC] expresses these properties far more explicitly. Moreover, as it was shown in [26], [27], LTL, CTL, μ-calculus, and PDL can be translated in FO^2[TC]. This fragment of 2-nd order logics is well-suited for model checking. As it follows from the results of [25], model checking problem for FO[TC] is NLOG-complete. The very structure of FO^2[TC] provides a possibility to evaluate it in straightforward manner on any finite model.

V. Model Building, Model Checking and Model Updating

The aim of runtime verification is to check the correctness of program behaviour in the course of program execution. In the framework of our per-flow abstract model of SDN this problem can be formalized as follows: given an initial network configuration Net_0, a list of PFP formal specifications $\Phi = \{\varphi_1, \ldots, \varphi_n\}$, and a sequence of network updating commands $\alpha = com_1, \ldots, com_i, \ldots$, check that for every $i,\ i \geq 1$, a network configuration $Net_i = com_i(Net_{i-1})$ satisfies all PFP specifications, i.e. all formulae from the list Φ are invariants of the sequence α.

To cope with this problem one needs some means to solve three separate tasks:

1) *model building*: given a network topology T and a SDN configuration Net build a formal model M_{Net};
2) *model checking*: given a formal model M_{Net} and a PFP specification φ check satisfiability $M_{Net} \models \varphi$;
3) *model updating*: given a formal model M_{Net} and a network updating command com build a a formal model $M_{com(Net)}$.

We briefly discuss our approach to the solution of these tasks.

A formal model of SDN configuration M_{Net} is completely specified by the finite relations R_{Net}, In, Out on the set of binary vectors (packet states and points). We use Binary Decision Diagrams (BDDs) to represent finite relations since BDDs are well-suited for set-theoretic manipulations with such relations (see [30]). Nowadays many software packages for computations on BDDs are available; in our project we used the toolset BuDDy due to its simple and convenient interface. This toolset provides ample means to solve the model building task. To this end it is sufficient to compute step by step in a straightforward way BDDs for all relations (PRC_r, R_r, and R_{tab}) involved in the definition of R_{Net} (see Section IV). It is worth noting that such BDDs for different switches can be computed independently and in parallel.

As for the second task, network model checking, it can be easily solved as well with the help of BDDs. Every formula φ from the specification language $\mathcal{L}$ is presented by an Abstract Syntax Tree (AST) T_φ. The leaves of this tree are variables X and Y, whereas the inner nodes of this tree are basic predicates R, I, O of $\mathcal{L}$, Boolean operators, quantifiers, and transitive closure operator TC. To check $M_{Net} \models \varphi$ it is sufficient to evaluate T_φ on a model M. Nodes marked with basic predicates invoke the corresponding BDDs (in some cases variable renaming may be required). If a node is marked with a Boolean operator or a quantifier then the corresponding procedures for manipulations with BDDs is used to assign a BDD to this node. The only type of node that needs some specific treatment are those marked with transitive closure operator TC. To build a BDD for $TC(R_0)$, given a BDD for a binary relation R_0, we use the following simple scheme: compute iteratively BDDs for relations

$$R_{i+1}(X,Y) = \exists Z\ (R_i(X,Z) \wedge R_i(Z,Y))$$

until $R_{i+1} = R_i$. This is the most time consuming stage of AST evaluation and much efforts have been made to implement it efficiently. Since every specification formula is closed, BDD assigned to the root of T_φ is a Boolean constant which indicates (un)satisfiability of φ on M.

Some heuristics are used to reduce the cost of AST evaluation. For example, in practice only some fields (VLAN, counters, etc.) in packet headers are subjected to SET_FIELD actions. Therefore, a packet header $\mathbf{h}$ may be split into two components $\mathbf{h} = (\mathbf{h}', \mathbf{h}'')$, where $\mathbf{h}'$ is composed of those bits that are not changed. Then 1-hop packet forwarding relation R_{Net} may be viewed as $R_{Net}(\mathbf{h}'_1, \mathbf{h}''_1, \mathbf{p}_1, \mathbf{w}_1, \mathbf{h}'_2, \mathbf{h}''_2, \mathbf{p}_2, \mathbf{w}_2)$. Such a presentation substantially reduces the size of BDDs.

An efficient solution of the third task — model updating — is crucial for the utility of runtime verification, since the performance of model updating procedure must be adequate to the rate of configuration updatings occurred in real-life networks. Therefore, the using of model building procedures for model modification is not the best solution. Actually, in some cases the basic relations in the SDN configuration models can be modified rather quickly. Since many important PFP requirements like the absence of loop-holes, reachability of certain end-points, etc. refer only to a transitive closure of 1-hop packet forwarding relation R^+_{Net}, it is urgent to modify efficiently this binary relation. Below we show how this can be done in the case when all rules in the flow tables have the same priority and a network configuration is updated by commands add and del that insert and delete packet forwarding rules.

To modify efficiently a formal model of SDN configuration M_{Net} at the execution of rule insertion command add the verifier must keep track of BDDs that represent the following binary relations on the set of packet states:

1) 1-hop packet forwarding relation $R_{Net}(\mathbf{s}, \mathbf{s}')$,

2) its transitive closure $R^+_{Net}(\mathbf{s}, \mathbf{s}')$,

3) packet forwarding relations for missed packets

$$R^{def}_{Net}(\mathbf{s}, \mathbf{s}') = \bigvee_{\mathbf{w} \in \mathcal{W}} R^0_{Net(\mathbf{w})}(\mathbf{s}, \mathbf{s}')$$

for all switches in the network.

Suppose that SDN controller sends an updating command $add(\mathbf{w}, r)$ via a control channel. Then our model modification procedure build BDDs for the following relations:

1) $\widehat{R}^{def}_{Net}(\mathbf{s}, \mathbf{s}') = R^{def}_{Net}(\mathbf{s}, \mathbf{s}') \wedge \neg PRC_r(\langle \mathbf{s}.hd, \mathbf{s}.pt \rangle)$

(the installation of a rule r reduces the domain of default instructions in the flow tables);

2) $\widehat{R}_{Net}(\mathbf{s}, \mathbf{s}') = (R_{Net}(\mathbf{s},\mathbf{s}') \wedge \neg (R^{def}_{Net}(\mathbf{s},\mathbf{s}') \wedge PRC_r(\mathbf{s}))) \vee$
$\vee\ \exists \mathbf{u}(R_r(\mathbf{s},\mathbf{u}) \wedge (T(\mathbf{u},\mathbf{s}') \vee (Out(\mathbf{u}) \wedge \mathbf{u} = \mathbf{s}')))$

(the installation of a rule r a) cancels the default instructions for those packets that fall into the scope of r, and b) extends the 1-hop packet forwarding relation to those packets that match the pattern of r);

3) $\widehat{R}^+_{Net}(\mathbf{s}, \mathbf{s}') = (\mathbf{s}' = \mathbf{w} \wedge R_1(\mathbf{s},\mathbf{s}')) \vee (\mathbf{s}' \neq \mathbf{w} \wedge R_2(\mathbf{s},\mathbf{s}'))$,

where

$R_1(\mathbf{s}, \mathbf{s}') = \exists \mathbf{u}((R^+_{Net}(\mathbf{s},\mathbf{u}) \vee \mathbf{s} = \mathbf{u}) \wedge \widehat{R}_{Net}(\mathbf{u},\mathbf{s}'))$,

$R_2(\mathbf{s}, \mathbf{s}') = \exists \mathbf{u}((R^+_{Net}(\mathbf{s},\mathbf{u}) \vee \mathbf{s} = \mathbf{u}) \wedge \exists \mathbf{v}(\widehat{R}_{Net}(\mathbf{u}, \mathbf{v}) \wedge$
$\wedge (R^+_{Net}(\mathbf{v},\mathbf{s}') \vee \mathbf{v} = \mathbf{s}')))$.

Theorem 1. *Suppose that Net is an arbitrary SDN configuration such that all forwarding rules have the same priority, and $com = add(\mathbf{w}, r)$ is command which installs a rule of the same priority. Then $R_{com(Net)} = \widehat{R}_{Net}$, $R^{def}_{com(Net)} = \widehat{R}^{def}_{Net}$.*

Moreover, if Net satisfies the following requirements expressed by FO^2 formulae:

$M_{Net} \models \forall X \forall Y (R^{def}_{Net}(X, Y) \rightarrow X.pt = octrl)$

(every instruction for missed packets directs these packets to a control output port);

$$M_{Net} \models \neg\exists X(X.sw = \mathbf{w} \wedge PRC_r(\langle X.hd, X.pt\rangle) \wedge$$
$$\wedge \exists Y(\widehat{R}_{Net}(X,Y) \wedge \exists X(R^+_{Net}(Y,X) \wedge$$
$$\wedge X.sw = \mathbf{w} \wedge PRC_r(\langle X.hd, X.pt\rangle)))))$$

(the installation of the rule r into the flow table of the switch $\mathbf{w}$ doesn't bring any loop-hole to the SDN configuration $com(Net)$).

then $R^+_{com(Net)} = \widehat{R}^+_{Net}$.

Theorem 1 suggests a time-saving way for computing a transitive closure of 1-hop packet forwarding relation for an updated configuration without referring to a iterative procedure. It should be emphasized that most network configurations appeared in practice satisfy both premises of Theorem 1. Usually, if a packet doesn't match any forwarding rule then either the packet should be drop, or a network manager should be notified about such packet. In OpenFlow protocol this is achieved by sending a PacketIn message to the SDN controller. As for the second requirement, loop-holes in network configurations are strongly undesirable and typical PFPs include this demand. Both premises are expressed by FO^2[TC] formulae and can be checked by our verification system.

To complete a picture of model modification techniques we show how to modify efficiently a SDN model M_{Net} when an updating command $com = del(\mathbf{w}, \langle \mathbf{z}, \mathbf{y}\rangle, \ell)$ disables some packet forwarding rules in the flow table of a switch $\mathbf{w}$. In this case we use two auxiliary packet state properties:

$$\Phi_{\mathbf{z},\mathbf{y},\mathbf{w}}(\mathbf{s}) = \bigvee_{i=1}^{N} E_{\mathbf{z}[i]}(\mathbf{s}.hd[i]) \wedge \bigvee_{j=1}^{k} E_{\mathbf{y}[j]}(\mathbf{s}.pt[j]) \wedge (\mathbf{s}.sw = \mathbf{w}),$$

$$\Psi^{pred}_{\mathbf{z},\mathbf{y},\mathbf{w}}(\mathbf{s}) = \exists \mathbf{s}'((R^+_{Net}(\mathbf{s},\mathbf{s}') \vee \mathbf{s} = \mathbf{s}') \wedge \Phi_{\mathbf{z},\mathbf{y},\mathbf{w}}(\mathbf{s}'))$$

A packet state $\mathbf{s}$ satisfies the first property iff it matches a forwarding rule to be deleted by com. The second property holds iff a packet can be routed from the state $\mathbf{s}$ in the configuration Net with the use of a rule to be deleted by com.

A model modification procedure builds BDDs for the following relations:

1) $\widehat{R}^{def}_{Net}(\mathbf{s},\mathbf{s}') = R^{def}_{Net}(\mathbf{s},\mathbf{s}') \vee (\Phi_{\mathbf{z},\mathbf{y},\mathbf{w}}(\mathbf{s}) \wedge$
$\wedge\, \mathbf{s}'.hd = \mathbf{s}.hd \wedge \mathbf{s}'.pt = octrl \wedge \mathbf{s}'.sw = \mathbf{w}),$

2) $\widehat{R}_{Net}(\mathbf{s},\mathbf{s}') = (R_{Net}(\mathbf{s},\mathbf{s}') \wedge \neg\Phi_{\mathbf{z},\mathbf{y},\mathbf{w}}(\mathbf{s})) \vee$
$\vee (R^{def}_{Net}(\mathbf{s},\mathbf{s}') \wedge \Phi_{\mathbf{z},\mathbf{y},\mathbf{w}}(\mathbf{s})),$

3) $\widehat{R}^+_{Net}(\mathbf{s},\mathbf{s}') = R_1(\mathbf{s},\mathbf{s}') \vee R_2(\mathbf{s},\mathbf{s}'),$

where

$$R_1(\mathbf{s},\mathbf{s}') = R^+_{Net}(\mathbf{s},\mathbf{s}') \wedge \neg(\Psi^{pred}_{\mathbf{z},\mathbf{y},\mathbf{w}}(\mathbf{s}) \wedge \Psi^{post}_{\mathbf{z},\mathbf{y},\mathbf{w}}(\mathbf{s}')),$$

$$R_2(\mathbf{s},\mathbf{s}') = \exists \mathbf{u}((R^+_{Net}(\mathbf{s},\mathbf{u}) \vee \mathbf{s} = \mathbf{u}) \wedge \Psi^{pred}_{\mathbf{z},\mathbf{y},\mathbf{w}}(\mathbf{u}) \wedge$$
$$\exists \mathbf{v}(\widehat{R}_{Net}(\mathbf{u},\mathbf{v}) \wedge \neg\Psi^{pred}_{\mathbf{z},\mathbf{y},\mathbf{w}}(\mathbf{v}) \wedge (R^+_{Net}(\mathbf{v},\mathbf{s}') \vee \mathbf{v} = \mathbf{s}'))).$$

Theorem 2. *Suppose that Net is an arbitrary SDN configuration such that all packet forwarding rules have the same priority ℓ, and a command $com = del(\mathbf{w}, \langle \mathbf{z}, \mathbf{y}\rangle, \ell)$ deletes rules of priority ℓ. Then $R_{com(Net)} = \widehat{R}_{Net}$, $R^{def}_{com(Net)} = \widehat{R}^{def}_{Net}$.*

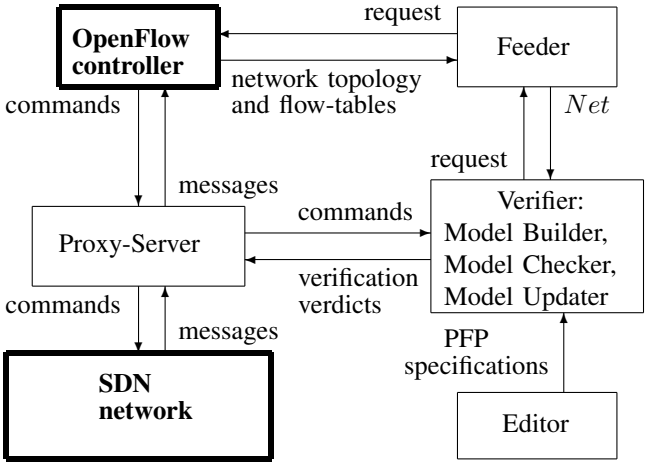

Fig. 1. The general arrangement of VERMONT

Moreover, if Net satisfies the requirement

$$M_{Net} \models \neg\exists X(\Phi_{\mathbf{z},\mathbf{y},\mathbf{w}}(X) \wedge \exists Y(R^+_{Net}(X,Y) \wedge \Phi_{\mathbf{z},\mathbf{y},\mathbf{w}}(Y)))$$

(packet forwarding rules deleted by the command com fire at most once along any route in the configuration Net)

then $R^+_{com(Net)} = \widehat{R}^+_{Net}$.

Theorems 1 and 2 show that whatever the size of a network and a number of forwarding rules in it might be, one needs to perform only a bounded number of operations on BDDs to modify a BDD representation of the SDN model in response to the execution of a network updating command. But these theorems are valid if all forwarding rules in network configurations have the same priority. A more involved techniques is implemented in VERMONT to achieve the same effect in a more general case for configurations which have packet forwarding rules with multiple priorities.

VI. A RUNTIME SDN VERIFICATION TOOLSET VERMONT: STRUCTURE AND FUNCTIONALITY

An SDN runtime verification toolset VERMONT includes four main components (see Fig. 1):

1) Proxy-Server: a module for intercepting OpenFlow commands and messages,

2) Verifier: a module for SDN model building, model checking and model updating,

3) Feeder: a module for supplying Verifier with data for building the initial SDN model,

4) Editor: a module for editing PFP specifications.

In the course of its operation VERMONT interacts with OpenFlow controller and SDN switches. When carrying out experiments we used an open source Floodlight SDN Controller [28] and Mininet — a software system for SDN prototyping [29] — instead of real SDNs.

One of the aims of network engineering is to provide such a loading of switches with forwarding rules as to guarantee compliance with the PFP. VERMONT provides some automation to the solution of this task. This toolset can be installed in line

with the control plane. It observes state changes of a network by intercepting messages sent by switches to the controller and command sent by the controller to switches. It builds an adequate formal model of a whole network and checks every event, such as installation, deletion, or modification of rules, port and switch up and down events, against a set formal requirements of PFP. Before a network updating command is sent to a switch VERMONT simulates its execution and checks whether an updated network configuration satisfies all requirements of current PFP. If this is the case then the command is delivered to the corresponding switch. Upon detecting a violation of PFP VERMONT may block the transmission of the updating command through the control channel, alerts a network administrator, and gives some additional information to elucidate a possible source of an error.

VERMONT may find vast applications in SDN technology. It can be attached to a SDN controller just to check basic safety properties (the absence of loops, blackholes, (un)reachability of certain hosts by specific routes, etc.) of the flow-tables managed by his controller. VERMONT may be also cooperated with software units (like FlowVisor) that aggregate several controllers. In this case VERMONT checks the compatibility of PFPs implemented by these controllers. This toolset can be also used as a fully automatic safeguard for every software application which implements certain PFP on a SDN controller. In this case VERMONT may handle individual network updating commands as well as the whole batches of such commands.

The principal functionality of the main modules of VERMONT are as follows.

Proxy-Server communicates with OpenFlow controller, SDN switches and Verifier. It intercepts all commands sent by the controller to SDN switches and all messages transmitted from the SDN switches to the controller. Proxy-Server is managed by the user of VERMONT (network manager) who can turn on and off this module, select its operational mode (SEAMLESS, MIRROR, INTERRUPT), set up and change the operation parameters. Depending on the chosen operation mode Proxy-Server may provide data (OpenFlow messages and commands) to Verifier, suspend some commands sent by the controller to SDN switches and block some of these commands by the results of their verification.

Verifier communicates with Proxy-Server, Feeder and Editor. This module runs three main algorithms:

- an initialization procedure which, given a description of a current network configuration (i.e. network topology and the content of all flow-tables in the network switches) Net_0, builds a BDD representation of a formal model M_{Net_0};
- a model checking procedure which verifies a set of PFP formal specifications $\Phi_1, \ldots, \Phi_n$ against a formal model of network configuration Net.
- a model updating procedure which, given a BDD representation of a formal model M_{Net} for a current network configuration Net and a network updating command com builds a BDD representation for the modified model $M_{com(Net)}$ for the updated SDN configuration $com(Net)$.

In the interaction with Proxy-server Verifier plays a role of server. It receives from Proxy-Server information on network updating, namely, a flow of OpenFlow network updating commands and messages on forwarding rules time-out expirations. Depending on the operation mode of the toolset it may check the formal models of updated configurations against the given PFP specifications and informs Proxy-server about the results of verification. In the interaction with Feeder Verifier plays a role of client. It may send requests to Feeder for the descriptions of a current network configuration. Verifier communicates with Editor as a server to receive formal specifications of a current PFP.

Feeder interacts with Verifier and with OpenFlow controller. At the requests from Verifier it communicates with the OpenFlow controller as a client and asks it about the necessary data on network topology and the content of flow tables. At receiving these data on current network configuration Feeder retransmits them to Verifier. We use Feeder as a separate module to make our system independent of a particular SDN controller.

By means of Editor a user of the Toolset may input PFP formal specifications, check their syntactic correctness, and send these specifications to Verifier.

The current version of VERMONT admits three operating modes.

1) In SEAMLESS mode VERMONT operates like a control flow channel between the OpenFlow controller and the network of switches. Proxy-Server does not invoke Verifier, commands from the Controller are not suspended and they are delivered to corresponding switches without delay. VERMONT proceeds to this mode either by the request from its user (manually), or at the shutting down of communication with Verifier (automatically).
2) In MIRROR mode Proxy-Server retransmits without delay all OpenFlow commands and messages to the corresponding parties (controller and switches) but the copies of these control flow data are delivered to Verifier. At receiving network updating commands Verifier checks their correctness; it informs a user about the results of the checking, but does not block incorrect commands.
3) In INTERRUPT mode VERMONT carries out a full-fledged runtime verification of network configurations and handle the flow of network updating commands sent to by the OpenFlow controller to SDN switches. All updating commands and statistics requests that depend on these commands are suspended by Proxy-Server until their verification is completed. The copies of suspended commands are delivered to Verifier. It simulates the execution of every such command on the current network configuration and checks the updated configuration against the PFP specifications received from Editor. If all PFP requirements are satisfied then Verifier allows Proxy-Server to sent the command to the corresponding switch. Otherwise, Verifier instructs Proxy-Server to drop the command and inform the network manager about this event.

	FT(60)	FT (80)	FT(100)	SUN-mod
number of rules	36900	49300	61500	15484
MB	2756	3689	4574	4714
MC Φ_1	0.4	0.4	0.4	51
MC Φ_2	30	36	32	–
MC Φ_3	–	–	–	222
MC Φ_3	–	–	–	316
MU(add) max	9	168	172	426
MU(add) min	3	3	3	1
MU(add) average	6	6	6	67
MU(del) max	178	174	176	307
MU(del) min	4	5	5	1
MU(del) average	8	9	10	99

time (ms)
MB — Model Building
MC — Model Checking
MU(add) — Model Updating by add command
MU(del) — Model Updating by del command

Table 1.

VII. Conclusions

To evaluate the performance of our runtime verification toolset VERMONT we made two series of experiments. In the first series we apply VERMONT to a SDN which has a fat-tree 3-level topology and includes 27 switches: 2 switches in the upper level, 5 pairs of switches in the middle level, and 5 triples of switches in the bottom level. The egress ports of the bottom-level switches are connected to H end-hosts, where H varies from 50 to 100. We denote this type of networks by FT(H). The length of packet header is $N = 40$.

An initial loading of flow tables enables every pair of end-hosts to communicate via a route of the shortest length. Each packet forwarding rule has a certain time-out. As soon as this time-out expires a rule is deleted from the table and the controller is notified about this event. At receiving this message the controller restores the table by sending command add to install the rule. VERMONT monitors message and command exchange between the controller and the network and checks the following two properties of SDN configurations: 1) Φ_1: there are no topological loops, and 2) Φ_2: some routes have length 5, but there are no routes of length 6.

In the second series of experiments we dealt with the model of Stanford University Backbone Net (SUN). This network has 16 switches, and each of them has three flow-tables. Stanford has made the entire configuration rule set public. The authors of [20] used in their research the description of a SUN configuration presented in [31], and we followed their example. The length of packet header in SUN is $N = 128$. In the case of SUN we used VERMONT for checking (along with Φ_1) two properties of SDN configurations: 3) Φ_3: there are no routes of length greater than 3, and 4) Φ_4: there are no routes of length greater than 4.

Tool	MB (ms)	MUC (ms)	Spec Lang
VERMONT (2014)	4700	50 – 700	FO^2[TC]
NetPlumber (2013) [20]	37000	2 – 1000	CTL
VeriFlow (2013) [19]	> 4000	68-100	Fixed properties
AP Verifier (2013) [21]	1000	0.1	Fixed properties
FlowChecker (2010) [11]	1200000	350 – 67000	CTL
Anteater (2011) [17]	400000	???	Fixed properties

MB — Model Building
MUC — Model Updating and Checking

Table 2.

The results of our experiments with networks of both types are presented in Table 1.

SUN is also used in many papers [11], [17], [19], [20], [21] on network verification as a benchmark. The results of comparative analysis of the performance of these tools is presented in Table 2. To make our comparative analysis fair we did not modify in this case SUN flow tables.

As it can be seen from Table 2, VERMONT has the most expressive PFP specification language and displays good performance in building initial models of SDN configurations. But some verification toolsets, such as VeriFlow [19] and AP-Verifier [21], overcome VERMONT in the efficiency of model updating. The high speed of both tools in model checking and model updating is due to specific presentation of packet state space $\mathcal{W}$. The set of packet headers $\mathcal{H}$ is divided into equivalence classes: headers $\mathbf{h}'$ and $\mathbf{h}''$ are equivalent w.r.t. a configuration Net if every switch identically processes packets with headers $\mathbf{h}'$ and $\mathbf{h}''$ when they arrive at the same port. This techniques makes it possible to build a succinct explicit graph-theoretic presentation of 1-hop packet forwarding relation R_{Net}. But such division of $\mathcal{H}$ into equivalence classes may be efficiently computed and modified only in the case when the packet forwarding rules don't change packet headers. Moreover, only a bounded class of basic network configuration properties can be checked for SDN models thus presented. VERMONT, in contrast, is able to work with arbitrary network configurations and it has far more expressive specification language. Nevertheless, we believe that the performance of our model updating procedures can be substantially improved with the help of new techniques similar to those used in [21]. This is one of the lines of our further research on this topic. Another important task to be solved is the designing and implementation of a new module for generating counter-example in those cases when a network configuration does not satisfy some PFP requirement expressed by a $\forall$-formula of PFP specification language.

REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, et al. Openflow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, v. 38, N 2, 2008, pp. 6974.

[2] OpenFlow Switch Specification. Version 1.4.0, October 14, 2013, https://www.opennetworking.org.

[3] H. Kim, N. Feamster. Improving network management with software defined networking. *Communications Magazine*, IEEE, 2013, p. 114-119.

[4] N. Foster, M. Harrison, M.J. Freedman, et al., Frenetic: A Network Programming Language. *Proc. of the 16th ACM SIGPLAN International Conference on Functional Programming*, 2011, p. 279-291.

[5] T. S. E. N. Zheng Cai, A. L. Cox. Maestro: A System for Scalable OpenFlow Control. Tech. Rep. TR10-08, Rice University, 2010.

[6] A. Voellmy, H. Kim, N. Feamster. Procera: A Language for High-Level Reactive Network Control. *Proc. of the First Workshop on Hot Topics in Software Defined Networks*, 2012, p. 43-48.

[7] A. Voellmy, P. Hudak. Nettle — a Language for Configuring Routing Networks. *Proc. of the IFIP TC 2 Working Conference on Domain-Specific Languages*, 2009, p. 211-235.

[8] R. W. Ritchey, P. Ammann. Using model checking to analyze network vulnerabilities. *Proc. of the International IEEE Symposium on Security and Privacy*, 2000, p. 156165.

[9] G. G. Xie, J. Zhan, et al., On static reachability analysis of ip networks. *Proc. of the 24-th Annual Joint Conference of the IEEE Computer and Communications Societies*, IEEE, 2005, v. 3, p. 21702183.

[10] T. Ridge, M. Norrish, P. Sewell. A rigorous approach to networking: TCP, from implementation to protocol to service. *Proc. of the International Conference on Formal Methods*, 2008, p. 294309.

[11] E. Al-Shaer, W. Marrero, A. El-Atawy, K. El Badawi. Network Configuration in a Box: Toward End-to-End Verification of Network Reachability and Security. *Proc. of the 17th IEEE International Conference on Network Protocols*, Princeton, New Jersey, USA, 2009.

[12] M. Canini, D. Venzano, et al. A nice way to test openflow applications. *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[13] M. Kuzniar, P. Perecini, M. Canini, et al. A SOFT way for OpenFlow switch interoperability testing. *Proc. of the 8-th international conference on Emerging networking experiments and technologies*, 2012, p. 265-276.

[14] D. Sethi, S. Narayana, S. Malik. Abstractions for Model Checking SDN Controllers. *Formal Methods in Computer Aided Design*, 2013.

[15] T. Ball, N. Bjorner, at al. Defined Networks. VeriCon: Towards Verifying Controller Programs in Software Defined Networks. *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, p. 282-293.

[16] E. Al-Shaer, S. Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. *Proc. of the 3-rd ACM Workshop on Assurable and Usable Security Configuration*, 2010, p. 3744.

[17] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, R.B. Godfrey, S.T. King. Debugging of the Data Plane with Anteater. *Proc. of the ACM SIGCOMM conference*, 2011, p. 290-301.

[18] P. Kazemian, G. Varghese, N. McKeown. Header space analysis: Static checking for networks. *Proc. of 9-th USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[19] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. *Proc. of 1-st International Conference "Hot Topics in Software Defined Networking" (HotSDN)*, 2012.

[20] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, S. Whyte. Real time network policy checking using header space analysis. *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[21] H. Yang, S. S. Lam, Real-time verification of network properties using atomic predicates. *Proc. of IEEE International Conference on Network Protocols*, 2013.

[22] Chemeritskiy E.V., Smelyansky R.L., Zakharov V.A. A Formal Model and Verification Problems for Software Defined Networks. *Proc. of the 4-th Workshop "Program Semantics, Specification and Verification: Theory and Applications"*, 2013, Yekaterinburg, Russia, p. 21-30.

[23] Chemeritskiy E.V., Zakharov V.A. On the Network Update Problem for Software Defined Networks. *Proc. of the 5-th Workshop "Program Semantics, Specification and Verification: Theory and Applications"*, Moscow, Russia, June 4, 2014, p. 26-37.

[24] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger. D. Walker. Abstractions for Network Update. *Proc. of ACM SIGCOMM*, 2012.

[25] Immerman N. Languages that capture complexity classes. *SIAM Journal of Computing*, v. 16, N 4, 1987, p. 760-778.

[26] Immerman N., Vardi M. Model checking and transitive closure logic. *Lecture Notes in Computer Science*, 1997, p. 291-302.

[27] Alechina N., Immerman N. Reachability logic: efficient fragment of transitive closure logic. *Logic Journal of IGPL*, 2000, v. 8, N 3, p. 325-337.

[28] Project FloodLight: Open Source Software for Building Software-Defined Networks: http://www.projectfloodlight.org/floodlight/

[29] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. *Proc. of the 9-th ACM Workshop on Hot Topics in Networks*, 2010.

[30] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677-691, 1986.

[31] Header Space Library and NetPlumber. https://bitbucket.org/peymank/hassel-public/.